Apple Assembly Line

$1.80

Volume 6 -- Issue 9 June, 1986

So Soon?

Another issue of Apple Assembly Line already? Well, readers sent in articles, Bob went on a writing binge, and we've managed to gain over a week in our efforts to get AAL back on schedule. You should all actually receive this issue during the month of June! One side effect of this acceleration is that Bill wasn't ready in time with the code to boot DOS 3.3 from his UniDisk 3.5. It looks like next month for that program and article.

What, Not Yet?

Osborne/McGraw-Hill reports that their copies of 65816 Assembly Language Programming, by Michael Fischer, arrived today (6/3), so our orders should be shipped within two weeks. We'll send them on to our customers just as soon as they arrive. Simon & Schuster has taken over all of Prentice-Hall's titles, so they are now the ones we are bugging about Programming the 65816, by David Eyes. The latest word from S & S is mid-July. Sigh.

We understand that there is a 65816 book from Sybex in the stores, but the people who have seen it aren't very impressed, describing it as a 6502 book with some '816 information gleaned from the data sheets but few examples.

More Disk Utilities

We are now carrying the highly-regarded disk utility package Copy II Plus. This includes disk and file copy programs, catalog and file handling utilities for both DOS and ProDOS, track and sector editing, and much more. List price for all this is only $39.95, but we'll have it for just $35 + shipping.

Using the 65816 Stack Relative Mode........Bob Sander-Cederlof

The 65802 and 65816 have two new address modes that allow you to reach into the stack. The "offset,S" mode lets you access position relative to the stack pointer, and the "(offset,S),Y" mode lets you access data indirectly through an address that is on the stack. The new address modes are available even when the 65802/16 is in the "emulation" mode.

The hardware adds the value of the offset to the current stack pointer to form an effective address. The stack pointer is always pointing one address below the end of the stack. Thus, an address of "1,S" points to the first item on the stack.

These new modes lead to interesting programming possibilities. When you design a subroutine, you have to decide how you are going to pass parameters into and out of the subroutine. Usually we try to use the A, X, and Y registers first. Another method puts the data or the address of the data after the JSR that calls the subroutine. ProDOS MLI calls use this method:

```
       JSR $BF00
       .DA #$C1,PARMS
```

In another method you push data or data addresses on the stack, and then call the subroutine. This is the preferred method in some computers, but not the 6502. The new modes make this mode work nicely in the 65802/16, though.

I coded up two examples to show how you can use the new modes, both message printing subroutines. The calling method requires telling the subroutine where to find a variable length message. In the first one (lines 1070-1330), I chose to push the address of the text on the stack before calling the printing routine. In the second example (lines 1340-1640), I used the method of storing the message text immediately after the JSR instruction.

Lines 1070-1110 print out two messages, using the first technique. I use the PEA (Push Effective Address) instruction to put the address of the first byte of the message text on the stack. This instruction pushes first the high byte, then the low byte, of the value of the operand. (I think I would prefer to have called it "PSH #value", because that is the effect. Then the PEI opcode, which pushes two bytes from the direct page, could be "PSH zp". But, nobody asked me.)

Anyway, let's look at the PRINT.IT subroutine. When the subroutine starts looking at the stack, it looks like this:

```
          |  msg addr lo  | 4,S
          | ------------- |
          |  msg addr hi  | 3,S
          | ------------- |
          |  ret addr lo  | 2,S
          | ------------- |
          |  ret addr hi  | 1,S
          | ------------- |
          |               |<---Stack Pointer
```

```
S-C Macro Assembler Version 2.0......DOS $100, ProDOS $100, both for $120
Version 2.0 DOS Upgrade Kit for 1.0/1.1/1.2 owners...................$20
ProDOS Upgrade Kit for Version 2.0 DOS owners........................$30
Source Code of S-C Macro 2.0 (DOS only)..................additional $100
Full Screen Editor for S-C Macro (with complete source code)..........$49
S-C Cross Reference Utility......without source code $20, with source $50
RAK-Ware DISASM.................without source code $30, with source $50
S-C Word Processor (with complete source code).......................$50
DP18 Source and Object...............................................$50
Double Precision Floating Point for Applesoft (with source code)......$50
ES-CAPE (Extended S-C Applesoft Program Editor)..........
                        Including Version 2.0 With Source Code..........$50
ES-CAPE Version 2.0 and Source Code Update (for Registered Owners)....$30
Bag of Tricks 2 (Quality Software)......................($49.95)  $45 *
Copy II Plus (Central Point Software)...................($39.95)  $35 *
Applesoft Toolbox Series (Roger Wagner Publishing).....each ($39.95)  $36 *
                                           all four ($159.80) $140 *
MacASM -- Macro Assembler for MacIntosh (Mainstay).........($150.00) $100 *
S-C Documentor (complete commented source code of Applesoft ROMs).....$50
Cross Assemblers for owners of S-C Macro Assembler.....$32.50 to $50 each
   (Available:  6800/1/2, 6301, 6805, 6809, 68000, Z-80, Z-8, 8048,
                8051, 8085, 1802/4/5, PDP-11, GI1650/70, others)

AAL Quarterly Disks.........................each $15, or any four for $45
 Each disk contains the source code from three issues of AAL,
 saving you lots of typing and testing.
 The quarters are Jan-Mar, Apr-Jun, Jul-Sep, and Oct-Dec.
(All source code is formatted for S-C Macro Assembler.  Other assemblers
require some effort to convert file type and edit directives.)

Diskettes (with hub rings)........................ package of 20 for $20 *
Vinyl disk pages, 6"x8.5", hold two disks each.................10 for $6 *
Diskette Mailing Protectors (hold 1 or 2 disks).............40 cents each
   (Cardboard folders designed to fit 6"X9" Envelopes.)    or $25 per 100 *
Envelopes for Diskette Mailers............................  6 cents each

65802 Microprocessor (Western Design Center)..................($95)  $50 *
quikLoader EPROM System (SCRG)...............................($179) $170 *
PROmGRAMER (SCRG).........................................($149.50) $140 *
Switch-a-Slot (SCRG)......................................($179.50) $170 *

"65816/65802 Assembly Language Programming", Fischer........($19.95)  $18 *
"Programming the 65816", Eyes...............................($22.95)  $21 *
"Apple //e Reference Manual", Apple Computer................($24.95)  $23 *
"Apple //c Reference Manual", Apple Computer................($24.95)  $23 *
"ProDOS Technical Reference Manual", Apple Computer.........($29.95)  $27 *
"Now That You Know Apple Assembly Language...", Gilder......($19.95)  $18 *
"Apple ProDOS:  Advanced Features for Programmers", Little..($17.95)  $17 *
"Inside the Apple //c", Little..............................($19.95)  $18 *
"Inside the Apple //e", Little..............................($19.95)  $18 *
"Apple II+/IIe Troubleshooting & Repair Guide", Brenner.....($19.95)  $18 *
"Apple ][ Circuit Description", Gayler......................($22.95)  $21 *
"Understanding the Apple II", Sather........................($22.95)  $21 *
"Understanding the Apple //e", Sather.......................($24.95)  $23 *
"Enhancing Your Apple II, vol. 1", Lancaster................($15.95)  $15 *
"Enhancing Your Apple II, vol. 2", Lancaster................($17.95)  $17 *
"Assembly Cookbook for the Apple II/IIe", Lancaster.........($21.95)  $20 *
"Beneath Apple DOS", Worth & Lechner........................($19.95)  $18 *
"Beneath Apple ProDOS", Worth & Lechner.....................($19.95)  $18 *
"Real Time Programming -- Neglected Topics", Foster..........($9.95)   $9 *
"Microcomputer Graphics", Myers.............................($14.95)  $14 *
"Assem. Language for Applesoft Programmers", Finley & Myers.($18.95)  $18 *
"Assembly Lines -- the Book", Wagner........................($19.95)  $18 *
"AppleVisions", Bishop & Grossberger........................($39.95)  $36 *

           * On these items add $2.00 for the first item and
             $.75 for each additional item for US shipping.
             Foreign customers inquire for postage needed.
       Texas residents please add 6 1/8 % sales tax to all orders.

        *** S-C SOFTWARE, P. O. BOX 280300, Dallas, TX 75228 ***
        ***                 (214) 324-2050                   ***
        *** Master Card, VISA, Discover and American Express ***
```

The LDA (3,S),Y instruction at line 1240 takes the address at 3,S and 4,S (which is the address of the first byte of the message) and adds the Y-register to it; then the LDA opcode picks up the message byte. After printing all the message and finding the terminating 00 byte, lines 1290-1320 move the return address up two slots higher in the stack (over the top of the message address). At the same time, the original copy of the return address is removed from the stack. Then a simple RTS takes us back to the caller, with a clean stack.

The second example uses a "message buried in the code" method. When PRINT.MSG looks at the stack, only the return address is there. The return address points to the third byte of the JSR instruction, one byte before the message text. Therefore the printing loop in lines 1500-1550 starts with Y=1. Lines 1560-1620 add the message length to the return address, so that an RTS opcode will return to the caller just past the message.

```
                        1000 *SAVE S.TEST 65816 CALLING SEQUENCES
                        1010 *--------------------------------
                        1020        .OP 65816
                        1030 *--------------------------------
                        1040 *      PEA address of message text
                        1050 *      JSR PRINT.IT
                        1060 *--------------------------------
000800- F4 0D 08        1070 T1     PEA MESSAGE.1
000803- 20 29 08        1080        JSR PRINT.IT
000806- F4 1B 08        1090        PEA MESSAGE.2
000809- 20 29 08        1100        JSR PRINT.IT
00080C- 60              1110        RTS
                        1120 *--------------------------------
                        1130 MESSAGE.1
00080D- 8D              1140        .HS 8D
00080E- CD C5 D3 D3
000812- C1 C7 C5 A0
000816- CF CE C5        1150        .AS -/MESSAGE ONE/
000819- 8D 00           1160        .HS 8D.00
                        1170 MESSAGE.2
00081B- 8D              1180        .HS 8D
00081C- CD C5 D3 D3
000820- C1 C7 C5 A0
000824- D4 D7 CF        1190        .AS -/MESSAGE TWO/
000827- 8D 00           1200        .HS 8D.00
                        1210 *--------------------------------
                        1220 PRINT.IT
000829- A0 00           1230        LDY #0        STARTING INDEX
00082B- B3 03           1240 .1     LDA (3,S),Y   NEXT CHARACTER OF MESSAGE
00082D- F0 06           1250        BEQ .2        ...TERMINATING $00
00082F- 20 ED FD        1260        JSR $FDED     PRINT THE CHAR
000832- C8              1270        INY
000833- D0 F6           1280        BNE .1        ...ALWAYS
000835- 68              1290 .2     PLA           MOVE RETURN ADDRESS
000836- 83 02           1300        STA 2,S       OVER THE TOP OF THE
000838- 68              1310        PLA           MESSAGE ADDRESS, PRUNING
000839- 83 02           1320        STA 2,S       THE STACK
00083B- 60              1330        RTS
                        1340 *--------------------------------
                        1350 *      JSR PRINT.MSG
                        1360 *      text of message, terminating zero
                        1370 *--------------------------------
                        1380 T2
00083C- 20 69 08        1390        JSR PRINT.MSG
00083F- 8D              1400        .HS 8D
000840- CD C5 D3 D3
000844- C1 C7 C5 A0
000848- C1 C6 D4 C5
00084C- D2 A0 CA D3
000850- D2              1410        .AS -/MESSAGE AFTER JSR/
000851- 8D 00           1420        .HS 8D.00
000853- 20 69 08        1430        JSR PRINT.MSG
000856- 8D              1440        .HS 8D
```

```
000857- C1 CE CF D4
00085B- C8 C5 D2 A0
00085F- CD C5 D3 D3
000863- C1 C7 C5    1450        .AS -/ANOTHER MESSAGE/
000866- 8D 00       1460        .HS 8D.00
000868- 60          1470        RTS
                    1480 *--------------------------------
                    1490 PRINT.MSG
000869- A0 01       1500        LDY #1        POINT TO FIRST CHAR
00086B- B3 01       1510 .1     LDA (1,S),Y   GET NEXT CHAR
00086D- F0 06       1520        BEQ .2        ...TERMINATING $00
00086F- 20 ED FD    1530        JSR $FDED     PRINT THE CHAR
000872- C8          1540        INY
000873- D0 F6       1550        BNE .1        ...ALWAYS
000875- 98          1560 .2     TYA           ADJUST THE RETURN ADDRESS
000876- 18          1570        CLC           BY ADDING THE MESSAGE LENGTH
000877- 63 01       1580        ADC 1,S
000879- 83 01       1590        STA 1,S
00087B- A9 00       1600        LDA #0        THE HIGH BYTE TOO
00087D- 63 02       1610        ADC 2,S
00087F- 83 02       1620        STA 2,S
000881- 60          1630        RTS           RETURN TO CALLER
                    1640 *--------------------------------
```

It might be instructive to look at how these two examples could be code in a plain 6502 environment. First, we must replace the PEA opcodes in lines 1070 and 1090 with the following:

```
        LDA #MESSAGE
        PHA
        LDA /MESSAGE
        PHA
```

Then PRINT.IT would require using temporary memory somewhere or writing self-modifying code. With a pointer in page zero, it could work like this:

```
00-             1250 RETURN.SAVE  .EQ $00,01
02-             1260 PNTR         .EQ $02,03
                1270 PRINT.IT
082F- 68        1280        PLA           POP RETURN ADDRESS
0830- 85 01     1290        STA RETURN.SAVE+1
0832- 68        1300        PLA
0833- 85 00     1310        STA RETURN.SAVE
0835- 68        1320        PLA           POP MESSAGE ADDRESS
0836- 85 03     1330        STA PNTR+1
0838- 68        1340        PLA
0839- 85 02     1350        STA PNTR
083B- A0 00     1360        LDY #0        STARTING INDEX
083D- B1 02     1370 .1     LDA (PNTR),Y   NEXT CHARACTER OF MESSAGE
083F- F0 06     1380        BEQ .2        ...TERMINATING $00
0841- 20 ED FD  1390        JSR $FDED     PRINT THE CHAR
0844- C8        1400        INY
0845- D0 F6     1410        BNE .1        ...ALWAYS
0847- A5 00     1420 .2     LDA RETURN.SAVE
0849- 48        1430        PHA           RELOAD RETURN ADDRESS
084A- A5 01     1440        LDA RETURN.SAVE+1
084C- 48        1450        PHA
084D- 60        1460        RTS           RETURN TO CALLER
```

PRINT.MSG also can be written in pure 6502 code with either self-modifying code or a pointer in page zero. Here is the self-modifying version:

Continued on page 14

Fast 16x16 Multiply & Divide in 65802............John Butterill

Ottowa, Ontario

Recently I needed a 16-bit multiplication subroutine in my 65802-enhanced Apple II. Naturally, I needed one that was both fast and short. I referred back to the Jan 86 AAL, which contained several examples for the 65802. The one named FASTER caught my fancy because it seemed a good compromise between size and speed. Then I made some changes which I think significantly improve it.

I noted that when you ROR the low half of the product into the multiplier, you get a bit out. This bit remains in the carry. If the low-product and the multiplier share the same location, then you can ROL in the low-product bit and ROL out the multiplier bit at the same time, instead of loading and LSR-ing the multiplier. By not having to load the multiplier, the Accumulator is free to contain the high half of the product without saving and loading it each time around. The result is rather more compact, fitting into 35 bytes (FASTER took 42 bytes).

It is also faster. By my calculations, the best and worst cases take 335 and 383 cycles, respectively. This includes the JSR to call the subroutine and the RTS to get back.

At the expense of two more bytes, I can save nine more cycles: delete line 1240 and add the following:

```
1304     ROR
1305     ROR A
```

This avoids the 17th trip through the loop, whose only purpose was to roll-in the final bit of the product.

By the way, some assemblers use the syntax "ROR A" to rotate the contents of the A-register. The S-C Macro Assembler and some others use the syntax "ROR" with a blank operand field for that mode. Then "ROR A" means to rotate the contents of the variable named "A", as in my program. To avoid confusion, you might want to change the variable names, avoiding the name "A".

```
                     1000 *SAVE BUTTERILL'S MULTIPLY
                     1010 *--------------------------------
                     1020 * 16 BIT MULTIPLY FOR 65802
                     1030 * MULTIPLIES A BY B
                     1040 * LEAVES ANSWER IN A & B
                     1050 *--------------------------------
00-                  1060 A       .EQ 0,1     MULTIPLIER, PRODUCT-LO
02-                  1070 B       .EQ 2,3     MULTIPLICAND, PRODUCT-HI
                     1080 *--------------------------------
                     1090 *   TIMING:  B=$0000 -- 27 CYCLES
                     1100 *            A=$0000 -- 335 CYCLES
                     1110 *            A=$FFFF -- 383 CYCLES
                     1120 *   (INCLUDING JSR AND RTS)
                     1130 *--------------------------------
                          1140        .OP 65802
                          1150 MULT16
000800- 18                1160        CLC          ENTER FROM 6502
000801- FB                1170        XCE
000802- C2 20             1180        REP #$20
000804- A5 02             1190        LDA B        IF B ZERO,
000806- F0 17             1200        BEQ .90      THEN BY-PASS
000808- C6 02             1210        DEC B
```

```
00080A- A9 00 00    1220        LDA #$0000
00080D- A2 10       1230        LDX #16         FOR 16 BITS
00080F- 18          1240        CLC             FOR 17'TH CYCLE
000810- 6A          1250 .10    ROR             ROLL OUT PRODUCT BIT
000811- 66 00       1260        ROR A           ROLL IN 'PLIER BIT
000813- 90 02       1270        BCC .20
000815- 65 02       1280        ADC B
000817- CA          1290 .20    DEX
000818- 10 F6       1300        BPL .10         CYCLES 17 TIMES
00081A- 85 02       1310        STA B
00081C- 38          1320 .30    SEC             EXIT TO 6502
00081D- FB          1330        XCE
00081E- 60          1340        RTS
00081F- 85 00       1350 .90    STA A           PROCEDURE FOR B=0
000821- 80 F9       1360        BRA .30
                    1370 *-------------------------------
```

A 16-bit by 16-bit division seems inherently messier. First, the divisor must be shifted left until it is at least greater than half the dividend. One can do a fast cycle which shifts the divisor all the way to the left, but for every shift left in this loop, the divisor must be shifted right again in the second (subtracting) loop.

In practice, I feel that the values would not be randomly distributed, but would be biased toward smaller values. I'm more likely to divide by 7 than by 32973, for example. Therefore it is worthwhile putting in the extra code to shift left only as far as is necessary. The scaling portion in my subroutine, lines 1240-1300, shift the divisor until either bit 15 = 1 or the divisor equals/exceeds the dividend.

In the second loop, lines 1310-1400, the shifted divisor is repeatedly compared to the dividend. If it is smaller, it is subtracted and a 1-bit goes into the quotient; otherwise a 0-bit goes in. The loop stops after it has operated with the divisor shifted back to its original position. This is ordinary long division, in binary. The comparison-subtraction is performed from one to 16 times, depending on the values.

As I calculate it, the best case (dividend=divisor) takes 82 cycles. The worst case, which I think would be $FFFF/1, takes 676 cycles. The time is a function of the number of significant bits in the answer.

[John also wrote a nice demonstration driver for his subroutines, allowing you to enter two hexadecimal values and see the result in hexadecimal. The source code for the demo is included on the monthly/quarterly disk.]

```
                    1000 *SAVE BUTTERILL'S DIVIDE
                    1010 *-------------------------------
                    1020 * 16 BIT DIVIDE WITH REMAINDER
                    1030 * DIVIDE B BY A
                    1040 * LEAVES QUOTIENT IN B,
                    1050 *        REMAINDER IN A
                    1060 *-------------------------------
                    1070 *   TIMING:  A=$0000 -- 39 cycles
                    1080 *            B>$7FFF -- 71 or 74 cycles
                    1090 *            A=B     -- 82 cycles
                    1100 *        A=1,B=$FFFF -- 676 cycles
                    1110 *-------------------------------
00-                 1120 A      .EQ 0,1     DIVISOR, REMAINDER
02-                 1130 B      .EQ 2,3     DIVIDEND, QUOTIENT
                    1140 *-------------------------------
```

```
                        .OP 65802
               DIV16
000800- 18              CLC          ENTER FROM 6502
000801- FB              XCE          NATIVE MODE
000802- C2 20           REP #$20     A-REG 16 BITS
000804- A2 00           LDX #0       START SCALE CNTR
000806- A5 00           LDA A        GET DIVISOR
000808- F0 22           BEQ .90      ...ZERO DIVISOR
00080A- 30 0A           BMI .30      ...DIVISOR > $7FFF
               *---SCALE DIVISOR----------------
00080C- C5 02  .10      CMP B        ALIGN A TO LEFT
00080E- B0 04           BCS .20      UNTIL > B
000810- E8              INX             OR BIT 15 SET
000811- 0A              ASL          & COUNT IN X
000812- 10 F8           BPL .10
000814- 85 00  .20      STA A        SCALED DIVIDEND
               *---START SUBTRACTING------------
000816- A5 02  .30      LDA B        GET DIVIDEND
000818- 64 02           STZ B        CLEAR QUOTIENT
00081A- C5 00  .40      CMP A        REPEATED CONDITIONAL
00081C- 90 02           BCC .50       SUBTRACTION.
00081E- E5 00           SBC A
000820- 26 02  .50      ROL B        ROL IN 1 IF SUBT.
000822- 46 00           LSR A               0 IF NO SUBT.
000824- CA              DEX
000825- 10 F3           BPL .40
000827- 85 00           STA A        REMAINDER
               *---RETURN TO CALLER-------------
000829- 38     .60      SEC          EXIT TO 6502
00082A- FB              XCE
00082B- 60              RTS
               *---FOR X/0, GIVE 0,0 ANSWER-----
00082C- 85 02  .90      STA B        DIVISION BY ZERO
00082E- 80 F9           BRA .60
               *--------------------------------
```

The Real Story about DOS and BRUN........Bob Sander-Cederlof

I was wrong. Some of you were kind enough to point it out. John Butterill sent a letter, and others called (sorry, names forgotten). I said, in the January 1986 AAL, that the reason BRUNning programs from inside Applesoft programs often did not work was the fact that DOS used a JMP rather than a JSR to call your program.

The truth is that DOS does call your program with a JMP, but there is still a return address on the stack. The BRUN command processor itself was called with a JSR, in a way. At $A17A there is a JSR $A180. The routine at $A180 jumps to the BRUN processor. So when your program finishes it will return to $A17D, right after the JSR $A180. From there it goes to $9F83.

At $9F83, DOS will finally exit from doing the BRUN command. If MON C is on, the carriage return from the end of the BRUN command will be echoed at this time. This can put you into a loop, however, because the BRUN command re-installed the DOS hooks in the input and output vectors. When the DOS hooks are installed, any character input or output will enter DOS first. Since we are still, in effect, inside DOS, because of the BRUN, we get into a loop. DOS is not re-entrant, as John Butterill put it. The BRUN command processor does a JSR $A851, which re-installs the DOS hooks. If your program tries to do any character I/O through calls to $FDED (COUT) or $FD0C (RDKEY), and you start up your program by BRUNning it from inside an Applesoft program, you will get DOS into a loop. Or, even if your program does not do any I/O, if MONC is on DOS can still get into a loop.

I still think the easiest way to avoid this problem is to avoid using BRUN inside Applesoft programs. Use BLOAD and CALL instead. But sometimes you may want to use BRUN, because you do not know in advance where the CALL address would be. One way to allow I/O inside your own program even though it is to be BRUN from inside an Applesoft program is to disconnect or bypass the hooks. You could output characters by JSR $FDF0, for example. But that would always go to the screen, and you may have a printer or an 80-column card or a modem hooked in, so that isn't a real solution. Another way is to dis-install the DOS hooks, by doing a JSR $9EE0 or the equivalent. The code at $9EE0 does this:

```
        LDX #3
.1      LDA $AA53,X
        STA $36,X
        DEX
        BPL .1
        RTS
```

This unhooks DOS, but leaves any other I/O devices you have connected hooked in. After doing this step, your program can freely call COUT or RDKEY without DOS even knowing about it. You might also want to store a zero at $AA5E, to turn off MONC. Your program can terminate then by a JMP $3EA, which will restore the DOS hooks.

An alternative that seems to work is to save and restore the location where DOS saves the entering stack pointer. This is the culprit which causes the crippling loop. At $9FB6, just before returning to whoever entered DOS, the stack pointer gets reset to the value it had when DOS was entered. If you enter DOS while you are still in DOS, the first value is replaced with the second. Then the final return point is lost, and it is loop-city. Your program can save and restore $AA59, where the stack pointer is kept:

```
YOUR.PROGRAM
        LDA $AA59       save DOS stack pointer
        PHA
        LDA #0          turn off MON C
        STA $AA5E

...do all your stuff, including I/O

        PLA
        STA $AA59
        RTS
```

This method has the advantage that your program can issue its own DOS commands by printing them, the way you would from Applesoft. For example, the following program will work when BRUN from inside Applesoft.

```
        .OR $1000
        .TF B.SHOW OFF
DEMONSTRATE
        LDA $AA59
        PHA
        LDY #0          issue DOS CATALOG command
.1      LDA MSG,Y
        JSR $FDED
        INY
        CPY #MSGSZ
        BCC .1
        LDA #0
        STA $AA5E    "NOMON C"
        PLA
        STA $AA59
        RTS
MSG     .HS 8D.84
        .AS -/CATALOG/
        .HS 8D
MSGSZ   .EQ *-MSG

100 PRINT CHR$(4)"MONC"
110 PRINT CHR$(4)"BRUN B.SHOW OFF"
120 PRINT "FINISHED"
```

However, that program will not work correctly if you just type "BRUN B.SHOW OFF" from the command mode. You will get a syntax error after the catalog displays, because the catalog command is left in the input buffer incorrectly. Oh well!

Toggling Between Two Values....................Jan Eugenides

In the course of my job as Technical Editor for MicroSPARC, Inc. (the publishers of Nibble and Nibble Mac magazines), I am often called upon to modify programs that we are going to publish to make them compatible with configurations other than the one the author originally wrote for. Recently, I had to change a program to toggle between Drive 1 and Drive 3, rather than Drive 1 and Drive 2 as it was originally coded. Here is the original subroutine which toggled the drive number stored in a variable named CD:

```
TOGGLE.DRIVE
        LDA CD
        CMP #1
        BEQ .1
        LDA #1
        STA CD
        BNE .2
.1      INC CD
.2      RTS
CD      .BS 1
```

This code takes a total of 19 bytes, including the variable CD. My task was to exactly replace this routine with one which would toggle between 1 and 3 rather than 1 and 2. It had to use the same number of bytes, or less. It looks easy enough, but I couldn't come up with a solution. All my routines required one or two more bytes. I finally took the easy way out and patched it with a JMP to a free space near the end of the program, and put my code there. It works, but is there a shorter way?

Bob, you are the best code squeezer around, so I thought I'd give the problem to you. You'll undoubtedly come up with some sneaky code that does the trick in three bytes or less!

An Answer for Jan.......................Bob Sander-Cederlof

I don't know if I am the best code squeezer or not, but I can't squeeze it all the way to three bytes! My best attempt is nine bytes:

```
TOGGLE.DRIVE
        LDA #1
CD      .EQ *-1
        EOR #2
        STA CD
        RTS
```

In general, you can toggle back and forth between any two values by using the EOR instruction. The toggle constant is simply the exclusive-or of the two values. For example, to toggle back and forth between the values $A0 and $B2, I would use "EOR #$12".

My subroutine changes 1 to 3 and 3 to 1, as you requested.

However, it is not functionally identical to the original code. The original code did not store the variable CD inside an immediate-mode LDA, as I did. If that troubles you, simply change that line to "LDA CD" and add the line "CD .BS 1" at the end. The result takes ten bytes, still well under the limit.

The original code also always had the side-effect of setting carry status, so you might need to add a "SEC" instruction. I doubt it, because the original code would be very weird if it depended on this side-effect.

The original code not only changed 3 to 1, but also changed any other value not already 1 into 1. This is also probably not a necessary feature, because prior code should have made sure that we started with a valid drive number.

I came up with several other approaches to the problem, all of which are shorter than the original subroutine:

```
TOGGLE.DRIVE
        LSR CD      3 TO 1, OR 1 TO 0
        BNE .1      IT WAS 3 TO 1
        LDA #3      CHANGE 1 TO 3
        STA CD
.1      RTS

TOGGLE.DRIVE
        CLC
        LDA CD
        ADC #2      1 TO 3, OR 3 TO 5
        AND #3      5 TO 1
        RTS
```

None of these are particularly tricky or sneaky. In fact, the first and shortest one is the most straightforward. What would be tricky or sneaky is if the original author depended on the hidden side-effects in his subroutine.

Continued from page 5

```
                  1640 PRINT.MSG
087B- 68          1650        PLA              GET RETURN ADDRESS
087C- 8D 86 08    1660        STA .1+1         LO-BYTE
087F- 68          1670        PLA
0880- 8D 87 08    1680        STA .1+2         HI-BYTE
0883- A0 01       1690        LDY #1
0885- B9 99 99    1700 .1     LDA $9999,Y      ADDRESS FILLED IN
0888- F0 06       1710        BEQ .2           ...TERMINATING $00
088A- 20 ED FD    1720        JSR $FDED        PRINT THE CHAR
088D- C8          1730        INY
088E- D0 F5       1740        BNE .1           ...ALWAYS
0890- 98          1750 .2     TYA              ADJUST THE RETURN ADDRESS
0891- 18          1760        CLC              BY ADDING THE MESSAGE LENGTH
0892- 6D 86 08    1770        ADC .1+1
0895- A8          1780        TAY              SAVE LO BYTE FOR A WHILE
0896- A9 00       1790        LDA #0           THE HIGH BYTE TOO
0898- 6D 87 08    1800        ADC .1+2
089B- 48          1810        PHA
089C- 98          1820        TYA
089D- 48          1830        PHA
089E- 60          1840        RTS              RETURN TO CALLER
                  1850 *--------------------------------
```

Using Apple's Protocol Converter.........Bob Sander-Cederlof

The "Protocol Converter" is a firmware-controlled method of turning the //c disk port into a multi-drop peripheral bus able to support up to 127 external I/O devices. The bus connects devices which have enough intelligence: an "Integrated WOZ Machine" (IWM) chip, a 6502-type chip, RAM, and ROM. Data is transferred in a serial bit-stream at roughly 250,000 bits per second. So far, the only device anyone is building to run on the P/C bus is the Unidisk 3.5 from Apple.

As far as I have been able to determine, Apple's only published information about the protocol converter is in the Apple //c Technical Reference Manual, pages 114-142. The listing of the //c firmware in the same Manual also is informative. A preliminary document was available to developers, but most of the material is now given in the //c manual. Tom Weishaar ("Uncle DOS") promises a future article on the P/C in his "Open Apple" newsletter. (By the way, the June issue of "Open Apple" used the term "Smartport" as synonymous with "Protocol Converter".)

The Apple //e interface card for the UniDisk 3.5 also supports a "real" Protocol Converter. The Apple Memory Expansion Card, CirTech Flipster, and Applied Engineering RamFactor provide the same software interface with most of the features of the protocol converter for one I/O device (the memory card itself).

Apple briefly mentions the Protocol Converter in the Apple Memory Expansion Card manual (Appendix B, last paragraph), but warns against using it. They say "using the assembly-language protocol is fairly complicated". Nevertheless, a significant amount of the Apple firmware is used to implement the protocol converter features. It appears that someone inside Apple intends that the P/C will be included in the firmware of most future block-oriented devices. From a software stand-point, it could be used regardless of whether the actual hardware used the IWM-based bus, a SCSI bus, or no bus at all.

In order to use the protocol converter firmware, you need first to find it. The first step in finding it is to find which slot it is in. All of the cards with P/C firmware (so far) are also cards which control or emulate disk drives and have firmware supporting the ProDOS device driver protocol. Cards with ProDOS device driver firmware can be identified by four bytes: $Cs01 = $20, $Cs03 = $00, $Cs05 = $03, and $Cs07 = $00. The first three bytes in that list are the same for all disk drive controllers. The zero value at $Cs07 distinguishes it as a disk controller with protocol converter firmware.

The next step is to find the entry point in the firmware for protocol converter calls. The byte at $CsFF is the key. That byte is the offset in the firmware page for ProDOS calls. If $CsFF = $45, for example, ProDOS device driver calls would be "JSR $Cs45". To get the address of the protocol converter entry point, add 3 to the ProDOS entry point. In my example, "JSR $Cs48" would enter the protocol converter firmware. The actual value will probably be different for each kind of card, so you have to use software to find out what it is.

A program to find the slot and build the address of the protocol converter could look like this:

```
pcaddr  .eq $01,$02
find.pc lda #0
        sta pcaddr
        ldx #$C7   slot = 7 to 1 step -1
.1      stx pcaddr+1
        ldy #7
.2      lda (pcaddr),y  $Cs07,05,03,01
        cmp pc.sig,y
        beq .3
        dex
        cpx #$cl
        bcs .1     try next slot
        sec        signal could not find pc
        rts

.3      dey
        dey
        bpl .2
        lda (pcaddr),y   $CsFF
        adc #2     carry was set
        sta pcaddr
        rts        carry clear signals pc found

pc.sig  .HS FF.20.FF.00.FF.03.FF.00
```

Once you have the address of the protocol converter firmware, you call it in a manner similar to ProDOS MLI calls. You must plug the address of the protocol converter entry into a "JSR" instruction, which is followed by a one-byte command code and a two-byte address. The command code is a number from $00 to $09 which specifies which action you want the protocol converter to take. The address is the address of a parameter block, which provides additional information for processing the command, or a place for the information returned by the command. After the protocol converter has finished processing your command, it returns control to the next byte after the pointer to the parameter block. If carry is clear, there was no error. If carry is set, the A-register contains an error code.

Since my FIND.PC program left the address in two page zero locations, we could simply put a JMP opcode ($4C) in front of the address to make it into a JMP instruction. Then our calls to the protocol converter would look like this:

```
callpc  .eq $00         (just before pcaddr)
        jsr find.pc
        bcs ...         ...no pc found
        lda #$4C        JMP opcode
        sta callpc
        ...      ...other code
        jsr callpc
        .da #cmd,parameters
        ...      ...more code
```

Apple warns programmers NOT to use any page zero locations when calling the protocol converter firmware, saying that some page

zero locations are used by that firmware. They do not say what locations they use, but my investigations show that they use bytes in the range from $40 to $4F. What they do is push those on the stack, put in their own data, and at the end restore the original contents from the stack. They use an awful lot of stack, up to 35 bytes. (The RamFactor firmware uses no more than 17 bytes of stack for protocol converter calls, including the two used by your JSR.) If you want be safe rather than possibly sorry, you can copy the PCADDR bytes up into your own program. You could even plug them into every JSR which calls protocol converter. A cleaner way might be like this:

```
                jsr find.pc
                bcs ...        ...no pc found
                lda pcaddr
                sta callp+1
                lda pcaddr+1
                sta callpc+2
                ...
                jsr callpc
                .da #cmd,parameters
                ...
        callpc  jmp *   address filled in
```

Description of Protocol Converter Commands

Apple defines ten commands for the protocol converter firmware. These are not necessarily identical in function for all devices which use the protocol converter. In fact, Apple's memory card uses two of the commands differently than the UniDisk 3.5 does. The protocol converter firmware in the RamFactor functions exactly the same as that in the Apple Memory Expansion Card.

The following chart summarizes the ten commands as implemented in the Apple Memory Expansion Card and RamFactor firmware. A more detailed description of each command follows the chart. I am particularly pointing this at the memory cards rather than the Unidisk 3.5, because I believe these cards will be more popular with hackers like you and me. Furthermore, the Unidisk 3.5 information is available in the //c manual, but Apple has not released this detail for owners of the memory card.

Parameters:	+0 cmd	+1 cnt	+2 unit	+3	+4	+5	+6	+7	+8
PC Status	$00	3	0	bufl	bufh	code			
RAM Status	$00	3	1	bufl	bufh	code			
Read Block	$01	3	1	bufl	bufh	blkh	blkm	blkl	
Write Block	$02	3	1	bufl	bufh	blkh	blkm	blkl	
Format	$03	1	1						
Control	$04	3	0/1	bufl	bufh	code			
Init	$05	1	0/1						
Read Bytes	$08	4	1	bufl	bufh	cnth	cntl	adrh	adrm	adrl
Write Bytes	$09	4	1	bufl	bufh	cnth	cntl	adrh	adrm	adrl

Error Codes	$01	Command not $00-$05,$08, or $09
	$04	Wrong parameter count
	$11	Invalid Unit Number
	$21	Invalid Status or Control code
	$2D	Block Number too large

PC Status (cmd $00, unit $00, code $00): reads the status of the protocol converter itself into your buffer. The status of a memory card is always 8 bytes, with the first byte = $01 and all the others = $00. Also returns with $08 in the X-register and $00 in the Y-register. ($0008 is the number of bytes stored in your buffer.) This is of value only for compatibility with other devices supporting protocol converter firmware.

RAM Status (cmd $00, unit $01, code $00 or $03): reads the status of the memory card into your buffer. Code $00 stores four bytes: the first is always $F8, and the other three are the number of blocks in the current partition (lo, mid, hi order). (Y,X) will equal ($00,$04) when it is finished, showing that four bytes were stored. Code $03 will store 25 bytes: the first four are the same as code $00 returned; the next 17 are the name of the card in "ProDOS Volume Name" format (length of name in first byte, ASCII characters of name with hi-bit off, padded with blanks); and finally, four zero bytes. The card name is "RAMCARD". (Y,X) will return ($00,$19) when finished, indicating that 25 bytes were stored.

Obviously, the Status commands will operate differently on a real P/C bus, and the actual details will vary according to the device you interrogate.

Read Block (cmd $01): reads the specified block from the memory card. (In RamFactor, the block number is relative, inside the currently selected RamFactor partition.) You can read a block into a buffer in //e Auxiliary Memory by calling the P/C with the RAMWRT soft-switch set to AuxMem.

Write Block (cmd $02): writes the specified block from your buffer into the memory card. (In RamFactor, the block number is relative, inside the current RamFactor partition.) If you are careful and follow all the rules, you can write a block from a buffer in Auxiliary Memory by calling the protocol converter with the RAMRD soft-switch set to AuxMem. You have to put the code that sets the RAMRD switch and calls the protocol converter, and its parameter block, in zero-page or stack-page motherboard RAM ($0000-01FF), or in the language card RAM area. Or, you can have both RAMRD and RAMWRT set for AuxMem and be executing a program from within AuxMem. I always have a conceptual battle dealing with this kind of bank switching.

Format (cmd $03): does nothing in a memory card.

Control (cmd $04): does nothing in a memory card. If the code is not $00, you get error code $21. The buffer is never used.

Init (cmd $05): does nothing in a memory card.

Open or Close (cmd $06 or $07): cause error code $01 in a memory card. These commands only apply to character-oriented devices, and memory is a block-oriented device (so says Apple). Maybe someday someone will build a peripheral which is character-oriented and includes P/C firmware.

Read Bytes (cmd $08): reads a specified number of bytes starting at a specified memory-card address into your buffer.

The byte count may be as high as $FFFF, but this would obviously wreak havoc inside your Apple. No checks are made inside the protocol firmware for reasonableness of the buffer address or the byte count, so be careful. You would NEVER read into a buffer in the I/O address range ($C000-$CFFF).

The memory-card address may be as high as $7FFFFF. (In RamFactor, the address is relative inside the current partition.) This corresponds to a total of 8 megabytes, which is only half the maximum capacity of a RamFactor card. Apple has arbitrarily limited us to this maximum, because they use the top bit of the card address to specify whether the buffer is in MainMem (bit 23 = 0) or AuxMem (bit 23 = 1). (Bit 23 of the address is bit 7 of the last byte of the parameter block.)

Write Bytes (cmd $09): writes a specified number of bytes from your buffer starting at a specified memory-card address. The details of byte count, buffer location, and memory-card address are the same as for the Read Bytes ($08) command.

The Unidisk 3.5 firmware interprets commands $08 and $09 differently. Unidisk uses this pair to read and write Macintosh disks, which have 524-byte blocks.

All of the RamFactor protocol converter commands operate within the current active partition. In the Apple card there is only one partition (the whole card). RamFactor has nine partitions, and you are always in one of them. If you start with a blank card, the first call to the RamFactor protocol converter will set up the first partition with all but 1024 bytes, make that partition the current active one, and empty all the others.

Bill Morgan's articles on interfacing the Unidisk 3.5 with DOS 3.3 illustrate the use of protocol converter calls with that device. The real power of the protocol converter concept will not be realized until a variety of devices are available which use it. Maybe its real future is bound up in the new 65816-based Apple //.

Generalized MLI System Error Handling.....Bob Sander-Cederlof

The ProDOS Machine Language Interface (MLI) returns an error code in the A-register if anything goes wrong. There are about 30 error codes, with values from $01 to $5A. BASIC.SYSTEM reduces the number of different error codes to 18, calling many of them simply "I/O ERROR". A nearly complete description of the error codes can be found in several references:

"Apple ProDOS--Advanced Features", pages 68-70.
"Beneath Apple ProDOS", pages 6-59 thru 6-61.
"ProDOS Technical Reference Manual", pages 77-79.

When I am working with a new program which has a lot of MLI calls, it is helpful to have one central error handler to print out the error information. Gary Little gives us such a subroutine on pages 66 and 67 of his "Apple ProDOS -- Advanced Features." Gary's program prints the message "MLI ERROR $xx OCCURRED AT LOCATION $yyyy", where xx is the hexadecimal error code and yyyy is the address of the next byte after the MLI call. You can mentally subtract 6 from the yyyy address to get the actual address of the JSR $BF00 that caused the error.

I assume you already know, if you are following me this far, that MLI calls take the form "JSR $BF00", followed by three data bytes. The first data byte is the opcode, and the other two are the address of the parameter block for the MLI call:

```
       JSR $BF00
       .DA #OPCODE,PARAMETERS
```

It would be nice if the general error handler would give us a little more information. First, I would like for it to print out the actual address of the JSR $BF00, rather than the return address. Second, I would like for it to print out the three bytes which follow the JSR $BF00.

First, I recoded Gary's routine so that it took a lot less space. (Littler than Little's!) I shortened the message and tightened the code. My version prints simply "AT" in place of "OCCURRED AT LOCATION." Then I used a message printing subroutine to print the two text strings, rather than the two separate loops he used. His took 83 bytes, mine only 56.

```
                  1000 *SAVE MLI.ERROR
                  1010 *--------------------------------
BF9C-             1020 CMDADR .EQ $BF9C
                  1030 *--------------------------------
F941-             1040 PRNTAX .EQ $F941
FD8E-             1050 CROUT  .EQ $FD8E
FDDA-             1060 PRBYTE .EQ $FDDA
FDED-             1070 COUT   .EQ $FDED
                  1080 *--------------------------------
                  1090 MLI.ERROR
0800- 48          1100        PHA          SAVE ERROR CODE
0801- A0 00       1110        LDY #QERR
0803- 20 1F 08    1120        JSR PRMSG
0806- 68          1130        PLA
0807- 20 DA FD    1140        JSR PRBYTE
080A- A0 0D       1150        LDY #QAT
080C- 20 1F 08    1160        JSR PRMSG
080F- AD 9D BF    1170        LDA CMDADR+1
0812- AE 9C BF    1180        LDX CMDADR
0815- 20 41 F9    1190        JSR PRNTAX
0818- 4C 8E FD    1200        JMP CROUT
```

```
                  1210 *-------------------------------
081B- 20 ED FD    1220 MSG1     JSR COUT
081E- C8          1230          INY
081F- B9 25 08    1240 PRMSG    LDA MSGS,Y
0822- D0 F7       1250          BNE MSG1
0824- 60          1260          RTS
                  1270 *-------------------------------
                  1280 MSGS
00-               1290 QERR     .EQ *-MSGS
0825- 8D          1300          .HS 8D
0826- CD CC C9
0829- A0 C5 D2
082C- D2 CF D2
082F- A0 A4       1310          .AS -/MLI ERROR $/
0831- 00          1320          .HS 00
0D-               1330 QAT      .EQ *-MSGS
0832- A0 C1 D4
0835- A0 A4       1340          .AS -/ AT $/
0837- 00          1350          .HS 00
                  1360 *-------------------------------
```

Next, I started adding the features I mentioned above. The final program takes 92 bytes, which is 9 more than Gary's. It displays the error message "MLI ERROR $xx AT $yyyy (op.addr)."

Lines 1080-1160 pick up the address MLI saved in the System Global Page, and sbtract six from it. The result is stored into the LDA $9999,Y instruction at line 1200. Horrors! Self-modifying code! The loop at lines 1180-1240 copies the three data bytes which follow the JSR $BF00 into the three variables at lines 1390-1410.

Lines 1260-1360 print out the error message. This loop differentiates between ASCII characters (bit 7 = 1) and data offsets (bit 7 = 0). The text to be printed is in lines 1430-1550. Note that I used the negative ASCII form for the text, and .DA lines for the data bytes which will be printed in hexadecimal. The expressions in those .DA lines compute an offset from the beginning of the subroutine, which will come out as a value less than $7F. I also used the value 00 to terminate the entire message. The $8D bytes are RETURN characters, to make sure the error message prints on a line by itself.

```
                  1000 *SAVE MLI.ERROR.PLUS
                  1010 *-------------------------------
BF9C-             1020 CMDADR .EQ $BF9C
                  1030 *-------------------------------
FDDA-             1040 PRBYTE .EQ $FDDA
FDED-             1050 COUT   .EQ $FDED
                  1060 *-------------------------------
                  1070 MLI.ERROR.PLUS
0800- 8D 3B 08    1080        STA ERRCOD   SAVE ERROR NUMBER
0803- AC 9D BF    1090        LDY CMDADR+1
0806- AD 9C BF    1100        LDA CMDADR   SUBTRACT 6 FROM ADDRESS
0809- 38          1110        SEC
080A- E9 06       1120        SBC #6
080C- 8D 1A 08    1130        STA CALADR+1      CALL ADDR LO
080F- B0 01       1140        BCS .1
0811- 88          1150        DEY
0812- 8C 1B 08    1160 .1     STY CALADR+2      CALL ADDR HI
                  1170 *-------------------------------
0815- A0 02       1180        LDY #2
0817- A2 03       1190        LDX #3       COPY OPCODE & PARMS ADDR
0819- BD 99 99    1200 CALADR LDA $9999,X       (ADDRESS FILLED IN)
081C- E8          1210        INX
081D- 99 3C 08    1220        STA PARMADR.H,Y
0820- 88          1230        DEY
0821- 10 F6       1240        BPL CALADR   ...UNTIL Y=-1
                  1250 *-------------------------------
```

```
0823- 30 03                 BMI .2          ...ALWAYS
0825- 20 ED FD         .1   JSR COUT
0828- C8               .2   INY
0829- B9 3F 08              LDA QERR,Y
082C- 30 F7                 BMI .1          ...ASCII CHAR
082E- D0 01                 BNE .3          ...DATA BYTE
0830- 60                    RTS             ...END
0831- AA               .3   TAX             USE AS INDEX
0832- BD 00 08              LDA MLI.ERROR.PLUS,X
0835- 20 DA FD              JSR PRBYTE
0838- 4C 28 08              JMP .2          NEXT CHAR
                       *--------------------------------
083B-                  ERRCOD      .BS 1
083C-                  PARMADR.H   .BS 1
083D-                  PARMADR.L   .BS 1
083E-                  OPCODE      .BS 1
                       *--------------------------------
083F- 8D               QERR    .HS 8D
0840- CD CC C9
0843- A0 C5 D2
0846- D2 CF D2
0849- A0 A4                 .AS -/MLI ERROR $/
084B- 3B                    .DA #ERRCOD-MLI.ERROR.PLUS
084C- A0 C1 D4
084F- A0 A4                 .AS -/ AT $/
0851- 1B                    .DA #CALADR-MLI.ERROR.PLUS+2
0852- 1A                    .DA #CALADR-MLI.ERROR.PLUS+1
0853- A0 A8                 .AS -/ (/
0855- 3E                    .DA #OPCODE-MLI.ERROR.PLUS
0856- AE                    .AS -/./
0857- 3C                    .DA #PARMADR.H-MLI.ERROR.PLUS
0858- 3D                    .DA #PARMADR.L-MLI.ERROR.PLUS
0859- A9                    .AS -/)/
085A- 8D 00                 .HS 8D.00
                       *--------------------------------
```

Practical Application of CRC....................Don Rindsberg

When I read Bob S-C's article on CRC in the February 1986 AAL, I said, "Very interesting, but who needs it". Well, it wasn't long before I ran into a real need myself!

I bought a used IBM PC-Jr and wanted to put my own routines in an auto-start ROM cartridge. After some sleuthing, I found that the power-up routine checks for signature bytes. If they are present, the routine checks the ROM's CRC, which must be $0000 or the machine locks up.

Not knowing the 65802 opcodes that Bob used, and being quite familiar with the 8088 language, I decided to translate the PC-Jr's CRC routine from "8088 dis-assembly language" to "plain vanilla 6502-ese". I simulated the 8088's registers with Apple RAM, and wrote subroutines for some of the 16-bit 8088 instructions.

Now here's what I think is strange about CRC's. If you pass all bytes of a set of data through the CRC generator and then the two CRC bytes themselves, the total CRC result is $0000! The PC-Jr add-on ROMs have the program in all except the last two bytes and the CRC of the program in those last two, so the total CRC for the entire ROM is $0000.

My 6502 code requires you to enter the start in Apple RAM and the length of the ROM data. For example, for a program starting at $2000 in Apple RAM, destined to be blown into a 2716 EPROM (2048 bytes), you would enter an address of $2000 and a length of $0800. These two values go into the first four bytes of the Apple zero page, so you can use a monitor instruction from inside the S-C Assembler like this:

```
       :$00:00 20 00 08
```

My program runs a CRC calculation on all but the last two bytes, and then prints out what the resulting CRC code is. If you store the CRC value in the last two bytes of the ROM image, add two to the length, and re-run my program, the result should be 0000. In a particular example with a 2716, it might look like this:

```
       :$00:00 20 00 08        (set up address & length )
       :$800G                  (run CRC calculation      )
       82DF                    (value of CRC computed    )
       :$20FE:82 DF            (store CRC in EPROM image)
       :$02:02                 (increase length by two   )
       :$800G                  (run CRC calcualtion      )
       0000                    (it worked!               )
```

My routines will not win the speed or elegance contests, but they give me the data!

If you want another check on your coding, run a CRC calculation on the Applesoft $D000 ROM with length $0800. You should get $D01E if you have an Apple II+ or original //e version. The enhanced //e gives a CRC of $3BD4 because of some small changes

Apple made.

By the way, I use my Apple to generate assembly language code for the IBM PC line. I created an 8086/8088 cross assembler based on the S-C Assembler for the purpose. Contact me if you need a tool like this: Don Rindsberg, The Bit Stop, 5958 S. Shenandoah, Mobile, Alabama 36608. Or call at (205) 342-1653.

```
                  1000 *SAVE ROM CRC CALCULATION
                  1010 *--------------------------------
00-               1020 LOCN   .EQ $00,01     ENTER DATA LOCN (L/H)
02-               1030 SIZE   .EQ $02,03     ENTER ROM SIZE (L/H)
04-               1040 AL     .EQ $04        SIMULATED 8088 REGISTERS
05-               1050 AH     .EQ $05
06-               1060 BL     .EQ $06
07-               1070 BH     .EQ $07
08-               1080 CL     .EQ $08
09-               1090 CH     .EQ $09
0A-               1100 DL     .EQ $0A
0B-               1110 DH     .EQ $0B
0C-               1120 PTR    .EQ $0C,0D     WORK POINTER
0E-               1130 CTR    .EQ $0E,0F     BYTE COUNTER
                  1140 *--------------------------------
F941-             1150 PRNTAX .EQ $F941
                  1160 *--------------------------------
                  1170        .OR $300
                  1180 *--------------------------------
0300- A5 00       1190 START  LDA LOCN       SETUP POINTER
0302- 85 0C       1200        STA PTR              TO ROM IMAGE
0304- A5 01       1210        LDA LOCN+1
0306- 85 0D       1220        STA PTR+1
                  1230 *--------------------------------
0308- 38          1240        SEC            GET BYTE COUNT - 2
0309- A5 02       1250        LDA SIZE
030B- E9 02       1260        SBC #2
030D- 85 0E       1270        STA CTR
030F- A5 03       1280        LDA SIZE+1
0311- E9 00       1290        SBC #0
0313- 85 0F       1300        STA CTR+1
                  1310 *--------------------------------
0315- A0 FF       1320        LDY #$FF       START CRC AT $FFFF
0317- 84 0A       1330        STY DL
0319- 84 0B       1340        STY DH
031B- C8          1350        INY            Y=0
031C- 84 05       1360        STY AH         INIT AH REG
                  1370 *--------------------------------
031E- B1 0C       1380 .1     LDA (PTR),Y    GET NEXT BYTE
0320- 20 3E 03    1390        JSR FOLD.BYTE.INTO.CRC
0323- E6 0C       1400        INC PTR        BUMP THE WORK POINTER
0325- D0 02       1410        BNE .2
0327- E6 0D       1420        INC PTR+1
0329- A5 0E       1430 .2     LDA CTR        DECREMENT THE BYTE COUNT
032B- D0 02       1440        BNE .3
032D- C6 0F       1450        DEC CTR+1
032F- C6 0E       1460 .3     DEC CTR
0331- A5 0E       1470        LDA CTR        TEST IF FINISHED
0333- 05 0F       1480        ORA CTR+1
0335- D0 E7       1490        BNE .1         ...KEEP GOING
0337- A6 0A       1500        LDX DL         DISPLAY THE RESULT
0339- A5 0B       1510        LDA DH
033B- 4C 41 F9    1520        JMP PRNTAX
                  1530 *--------------------------------
                  1540 FOLD.BYTE.INTO.CRC
033E- 45 0B       1550        EOR DH
0340- 85 0B       1560        STA DH
0342- 85 04       1570        STA AL
0344- 20 6E 03    1580        JSR ROLAX4     8088 "ROL AX,C"
0347- 20 98 03    1590        JSR EORAD      8088 "EOR DX,AX"
034A- 20 77 03    1600        JSR ROLAX1     8088 "ROL AX,1"
034D- A5 0B       1610        LDA DH         SWAP BYTES IN REG-D
034F- A6 0A       1620        LDX DL
0351- 86 0B       1630        STX DH
0353- 85 0A       1640        STA DL
0355- 20 98 03    1650        JSR EORAD      8088 "EOR DX,AX"
0358- 20 83 03    1660        JSR RORAX4     8088 "ROR AX,C"
```

```
035B- A5 04           LDA AL
035D- 29 E0           AND #$E0
035F- 85 04           STA AL
0361- 20 98 03        JSR EORAD     8088 "EOR DX,AX"
0364- 20 8C 03        JSR RORAX1    8088 "ROR AX,1"
0367- A5 04           LDA AL
0369- 45 0B           EOR DH
036B- 85 0B           STA DH
036D- 60              RTS
                 *--------------------------------
                 *   SIMULATE 8088 "ROL AX,C"
                 *--------------------------------
036E- 20 77 03   ROLAX4 JSR ROLAX1  SHIFT 4 BITS BY SHIFTING
0371- 20 77 03        JSR ROLAX1          1 BIT 4 TIMES
0374- 20 77 03        JSR ROLAX1
                 *--------------------------------
                 *   SIMULATE 8088 "ROL AX,1"
                 *--------------------------------
0377- A5 04      ROLAX1 LDA AL      8088 "ROL" SHIFTS END AROUND
0379- 0A              ASL           WITHOUT LEAVING A BIT IN CARRY
037A- 26 05           ROL AH
037C- 90 02           BCC .1        6502 DOES LEAVE A BIT IN CARRY,
037E- 09 01           ORA #$01      SO LETS MERGE CARRY IN HERE.
0380- 85 04      .1   STA AL
0382- 60              RTS
                 *--------------------------------
                 *   SIMULATE 8088 "ROR AX,C"
                 *--------------------------------
0383- 20 8C 03   RORAX4 JSR RORAX1  SHIFT 4 BITS BY SHIFTING
0386- 20 8C 03        JSR RORAX1          1 BIT 4 TIMES
0389- 20 8C 03        JSR RORAX1
                 *--------------------------------
                 *   SIMULATE 8088 "ROR AX,1"
                 *--------------------------------
038C- A5 05      RORAX1 LDA AH      8088 "ROR" SHIFTS END AROUND
038E- 4A              LSR           WITHOUT LEAVING A BIT IN CARRY
038F- 66 04           ROR AL
0391- 90 02           BCC .1        6502 DOES LEAVE A BIT IN CARRY,
0393- 09 80           ORA #$80      SO LETS MERGE CARRY IN HERE.
0395- 85 05      .1   STA AH
0397- 60              RTS
                 *--------------------------------
                 *   SIMULATE 8088 "EOR DX,AX"
                 *--------------------------------
0398- A5 04      EORAD LDA AL
039A- 45 0A           EOR DL
039C- 85 0A           STA DL
039E- A5 05           LDA AH
03A0- 45 0B           EOR DH
03A2- 85 0B           STA DH
03A4- 60              RTS
                 *--------------------------------
```

Apple Assembly Line is published monthly by S-C SOFTWARE CORPORATION, P.O. Box 280300, Dallas, Texas 75228. Phone (214) 324-2050. Subscription rate is $18 per year in the USA, sent Bulk Mail; add $3 for First Class postage in USA, Canada, and Mexico; add $14 postage for other countries. Back issues are available for $1.80 each (other countries add $1 per back issue for postage). A subscription to the newsletter and the Monthly Disk containing all source code is $64 per year in the US, Canada and Mexico, and $87 to other countries.